Oracle Application Development Framework

Angels in the Architecture

Chris Muir & Frank Nimphius
Oracle Corporation

keywords

ADF, Architecture, Patterns, JDeveloper

Introduction

In the rush to implement projects, programming teams often focus on the nitty-gritty development details to get the job done. Agile development promotes rapid iteration, with an importance on implementing small incremental feature enhancements each iteration. Yet it can often forget the bigger picture of design and architecture, allowing development teams to paint themselves into a corner with ill-conceived application architectures.

Oracle's Application Development Framework (ADF) is an extremely flexible application framework from an architecture perspective. ADF promotes reuse as a primary feature and offers to adopters all the benefits of reuse, through the concepts of ADF libraries, page templates, CSS skins, declarative components, business services and most importantly of all bounded task flows which are akin to web services but for web page development mapped around business processes.

But like many flexible solutions, it's left up to ADF developers to choose the best combination of features for their own use case. And this is where the world of project pressures and ADF systems may lose their way, as developers are initially focusing and learning the nitty-gritty programming details, by natural extension best practices and good architecture follow. Who is considering the bigger picture of how all those ADF reusable components come together? And what established patterns are there for ADF application architectures?

Prerequisite Knowledge

In order to understand the ADF architectural concepts put forth in this article you must be familiar with ADF basics, including ADF Business Components, ADF Faces and importantly task flows and ADF libraries. Arguably to architects and the like not familiar with ADF they may still draw some value from this article, in particular an appreciation of how ADF and the task flow provides a level of reuse and architectural flexibility not seen by other web frameworks.

ADF Architectural Patterns

An evolving series of videos by Oracle's ADF Product Management team (http://youtu.be/toEuQvp73h8) has established patterns for constructing ADF applications for ADF designers, architects and senior developers. These patterns while not a guaranteed blueprint for success, establish different practices with both positives and negatives that make them suited to different requirements, teams, skill sets, infrastructure and development budgets. Indeed ADF teams must consider each pattern against their own circumstances and needs to pick what will work for them.

While readers are encouraged to watch the previously linked video, this article will broadly cover each of the patterns covered by the presentation.

Pattern Genealogy

In considering the following proposed patterns it's useful to understand that the patterns do follow an evolutionary path or in other words could be considered to have a genealogy. As will be seen each pattern builds upon the successes or avoids weaknesses of its predecessors, and in some cases as you'll see with the Sum-of-the-Parts pattern this opens up a wealth of options for the patterns that follow it.

In studying the patterns readers need to be careful to not assume that the last pattern is the "ultimate" pattern that every ADF project should take on board. As described previously, each pattern has a number of positive and negatives that suit different requirements, teams, skill sets and development budgets. There is no "on size fits all".

The Small and Simple Architectural Pattern

In investigating ADF architectural patterns it helps to both start with a simple pattern for learning purposes, and as a mechanism for introducing the diagramming notation which readers will need to understand for later more sophisticated patterns covered in this article.

The *Small and Simple Architectural Pattern* as shown in Figure 1 is a pattern that should be familiar to most JDeveloper beginners, even those from the JDeveloper 10g past, as its basic structure of 1 workspace and a Model and ViewController project is what the default Fusion Web Application wizard creates by default:

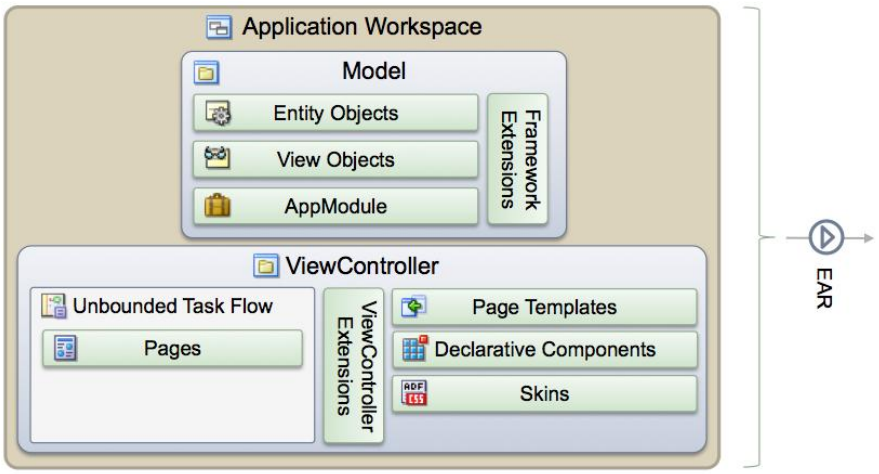

Figure 1 - Small and Simple Architecture Pattern

Beyond the single workspace containing the Model and ViewController projects, the other characteristics of the pattern are:

1. Contained with the Model project are the standard ADF Business Component (ADF BC) objects including Entity Objects (EOs), View Objects (VOs) an Application Module (AM) and recommended Framework Extension Classes.

2. In context of the ViewController project is contains only an Unbounded Task Flow (UTF) composed of pages, and some reusable structures such as page templates, declarative components, skins and associated utility classes.

3. In terms of deployment units, the workspace results in a single Java EE EAR file deployed to our Java EE server.

In essence this pattern is *Small and Simple* as it composed of so few architectural pieces as we'll see in later patterns, and is very familiar to many JDeveloper and ADF developers out there. From a design perspective focusing on architecture, there's not a lot to consider, possibly how many root ADF BC application modules should the application have, or a focus on a page by page design like traditional web site development, but nothing that would be considered a major stretch of an ADF architects skills.

This pattern presents a number of advantages including:

1. It's self contained – there are no external reusable components that must be built, versioned and deployed separately to the application.

2. As extension building and deploying the application is a simple one click affair. No additional infrastructure such as Continuous Integration (CI) servers like Hudson or build dependency tools such as Apache Maven are required (though they should still be considered as a best practice).

3. And ignoring the code and infrastructure, given its small and simple nature the solution suits small teams who don't have the resources to maintain larger complex systems, and beginners who don't want to be overwhelmed by architecture decisions when they're just learning the framework, and even small applications where a sophisticated architecture is just overkill.

However there are disadvantages of this pattern:

1. To keep this pattern simple, no reusable Bounded Task Flows (BTF) were introduced. The knock on affect is the lost opportunities that BTFs provide such as reusable page fragments and the associated reusable processes. Moving beyond the technical, without BTFs programmers lose the opportunity to talk to business analysts in the language of the business, "processes" and "flows", which BTFs articulate well.

2. Focusing on the code solution, as one or more developers are essentially working on one code base within one workspace, by inference it's very easy for programmers to create tightly coupled, poorly modularized code. Senior and disciplined programmers may be able to avoid this, but it would have been helpful that the architecture solution had provided a mechanism to separate concerns so programmers wouldn't have to think about best practices, the end pattern would have just inherently been loosely coupled, well modularized, easily unit tested and so on.

3. And obviously as all the code is packed into one workspace, as the solution grows, the build process is still an all or nothing affair. Change one little thing, you need to rebuild the whole application which can take considerable time.

The Colossal Architectural Pattern

In complete contrast the *Colossal Architectural Pattern* as the name suggests becomes overly large as seen in Figure 2:

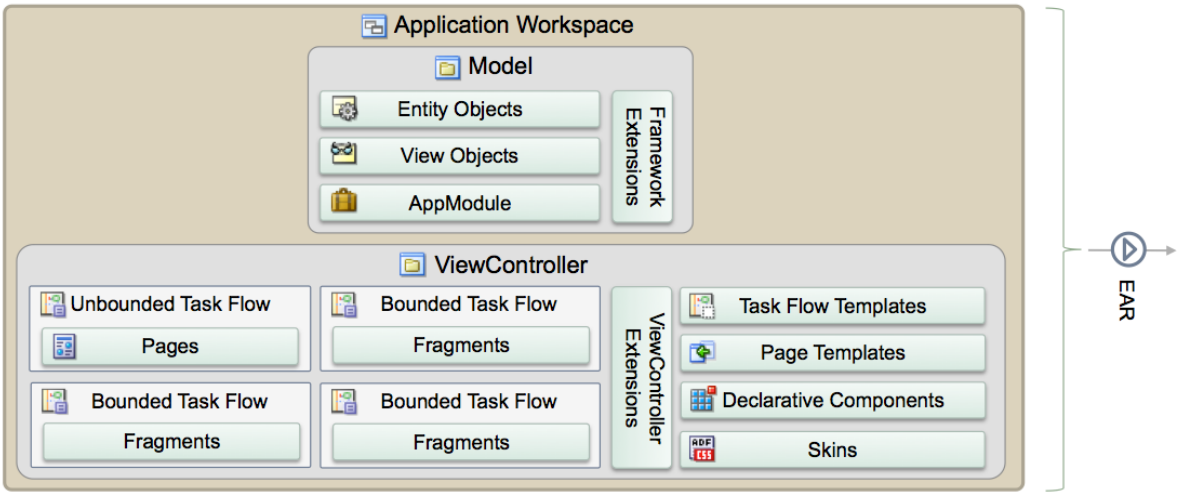

Figure 2 - The Colossal Pattern

A key change in the characteristics of this pattern over the previous is we introduce Bounded Task Flows (BTFs) for the first time, the number depending on the requirements and design of the system. For all intent and purposes the rest of pattern is the same with a single workspace, Model and ViewController projects, and a single EAR deployment profile. Potentially the only minor change is through the introduction of Bounded Task Flows, Task Flow Templates are now a viable option for reuse too.

In designing an application based on the Colossal Pattern and characteristics, the major design consideration is that purely of the Bounded Task Flows. BTFs like web services have many of the same design intricacies such as how granular should the task flows be? What should be their functional boundaries, how many should you have, do you just create fine grained single activity flows, or coarse grained multi activity flows, or a mixture of both? These are all choices for the ADF architect to make and enforce across the system to be developed.

Ultimately the many benefits of the Colossal Pattern are those introduced by Bounded Task Flows themselves:

1. The introduction of BTFs now allows a reusable architecture of pages and/or page fragments based around the processes the BTFs model. Previously if the same functionality or screen layout and data was wanted on multiple pages, it would have had to be coded twice. Now with BTFs such a construct can be defined once and consumed by two master pages, much like 3GL languages make use of functions for reuse.

2. Conversations with business analysts can move away from web concepts such as pages, HTML and CSS and can no move to discussions on processes, tasks, transactions and business outcomes, speaking the "language of the business" as you will and hopefully reducing the barrier between IT and the business.

3. As BTFs present an API, with a name, parameters both in and out, and a defined set of steps, the concept of designing by contract is now possible where requirement specifications, design and testing specifications can be written around each BTF.

4. Arguably another benefit for the Colossal Pattern is it is still a relatively simple architecture as it only has one workspace.

This simplistic architecture is potentially the downfall of the Colossal Pattern too:

1. While BTFs introduce a level of modularization into the application where different functions can be mapped to each BTF, hopefully promoting loose coupling and improved unit testing, it can't be guaranteed in the one workspace where developers work with each others' code.

2. As our application gets larger building the application is still an all or nothing affair.

3. If we broaden this out to testing, as we can't 100% guarantee code isn't accidentally coupled, we can't necessarily isolate parts of the solution to test standalone, we're just not sure if there's dependencies between code units. This further presents itself as a problem for regression testing where ideally we'd like to segment parts of our application so if a change occurs in one of those segments, we'd only need to retest that part. Because of the weak modularization we just can't guarantee the boundaries between the segments exist, so a regression test likely has to cover all of the application at much expense (particularly if regression testing is a manual rather than automated exercise).

4. And finally BTFs do inject their own complications into the software build. BTFs aren't just a reusable page or page fragment. They have sophisticated and what could be considered to beginners complex options around transactions and ADF data control scopes that require a medium level of ADF knowledge to implement and use correctly.

The Sum-of-the-Parts Architectural Pattern

So an undesirable part of the Colossal Pattern is the concerns around tight coupling and poor modularization. While senior and experienced developers may be able to ensure this doesn't occur, ideally it would be great if the ADF framework itself could promote a solution that makes all of this less likely to occur.

In the *Sum-of-the-Parts Pattern* this goal is realized through the implementation of ADF Libraries as shown in Figure 3:

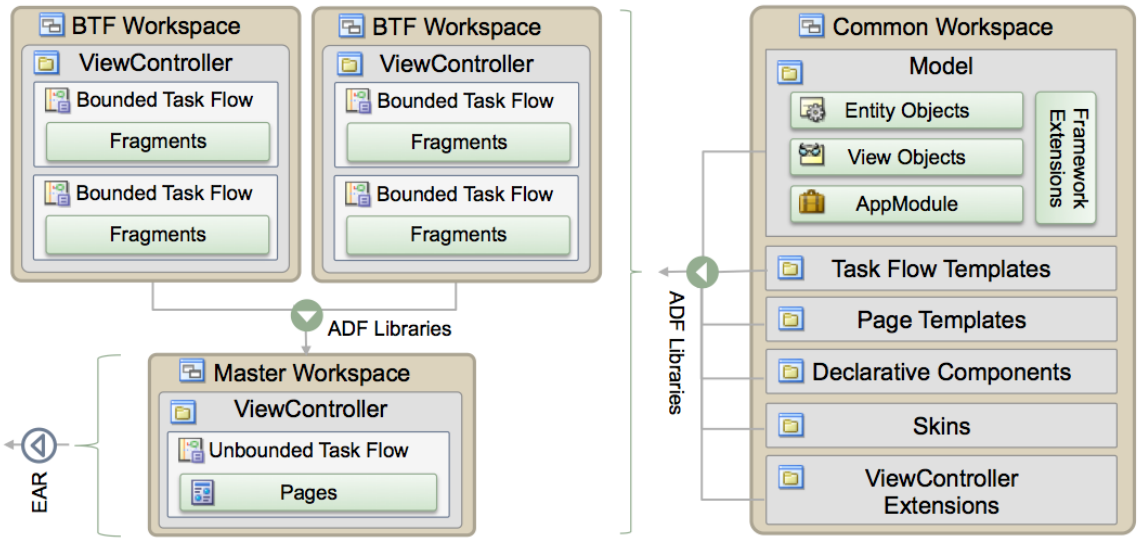

Figure 3 - The Sum-of-the-Parts Pattern

As ADF Libraries allow workspace projects to be packaged and consumed by other JDeveloper ADF workspace projects, containing ADF features such as ADF Business Components, Task Flow Templates, Page Templates, Declarative Components, and most importantly Bounded Task Flows, this allows us to break our logical application into many separate workspaces where developers can work in isolation from each other until they need to bring their work back together in a single composite application. Similar to previous the analogy here is of separate 3GL modules/classes with functions that different team members write, and other team members make use of.

From a design perspective the separated architecture allows Bounded Task Flows to be placed in 1 or more separate workspaces, consuming the more primitive reusable components, and consumed by a master/composite application. Of course this may sound simple but the architect is then hit with the challenge of what and how many BTFs should go in each BTF workspace, and the previous pattern's issue of how granular the BTFs should be isn't solved (and no pattern will solve this – this is something each ADF team will need to solve themselves with a goal of consistency being important).

This separated architecture presents many obvious benefits including:

1. The BTFs have an excellent separate of concerns, the opportunity to achieve loose coupling, and can be easily tested standalone as the separated workspaces clearly isolated BTFs in one workspace from another.

2. Previously for large development teams, all working on the one workspace can become a bit of a version control nightmare as developers are continuously having to update their local workspace from updates checked into the version control repository. With the new separate architecture developers can be aligned with their own workspace and live a level of isolation from other developers' work (though of course eventually all the code must make use of each other, so there is no ultimate isolation here though this would never be desirable as why would you want to write code that nobody else uses?).

3. And ideally developers now have ownership of code. Where previously it was quite possible that code was all mashed together in one workspace making identification of who was responsible for a bug harder, now with the clear architectural separation if a BTF workspace fails a round of unit testing, the developers responsible is much more obvious. From a psychological perspective this will make developers much more attached to their code and quality will rise as a result. Academically it's curious here how a technical feature of ADF can have an impact on your teams' psychology.

But with all this flexibility comes a cost. This is a relatively complex architecture compared to the previous patterns. As a new large issue comes to the fore. You'll now need to build your workspaces in a specific order, from the least dependent to the most in correct dependent order. This will require tools to track the dependencies, tools to track the required versions of the dependents and dependee, and skills amongst your teams to deal with some of the challenging build issues that all software development teams get into in this arena. This is the overhead of large software development projects that is often underestimated. At many organizations they have change control teams who are responsible for this work alone.

The Two-for-One Architectural Pattern

As can be seen the Sum-of-the-Parts pattern opens up many interesting architectural options in terms of ADF application development, a level of flexibility in application design that isn't afforded in many other web frameworks, realized through the ADF Library and Bounded Task Flow.

Of particular interest is one advantage that wasn't covered in the Sum-of-the-Parts pattern. If BTFs can be turned into ADF Libraries, and consumed by the main application's workspace, couldn't in theory they be consumed by other master application workspace too? Isn't it likely that building say a BTF to show Customer Details will be a common requirement across the systems of an enterprise for example?

Indeed this is one of the major benefits of Oracle's framework, that you can build BTFs to be reused across a wide range of applications with the opportunity to dramatically cut your overall development time by reusing previously created ADF artefacts.

The *Two-for-One Architectural Pattern* introduced in Figure 4 from an architectural perspective shows the sharing of BTFs via BTF workspaces and ADF Libraries across multiple master workspaces:

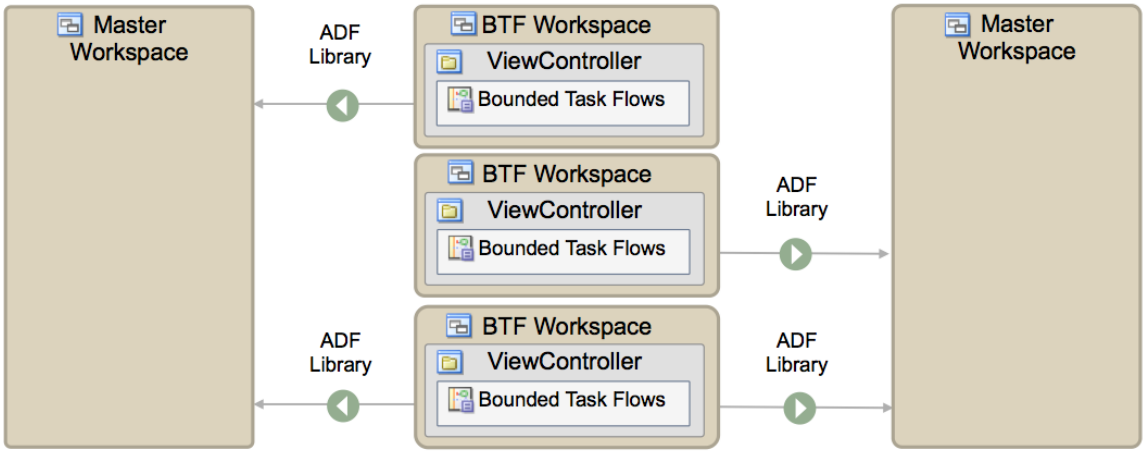

Figure 4 - The Two-for-One Pattern

This level of reuse is certainly an exciting opportunity for development teams and could be considered a nirvana not experienced in web development previously. "Reuse like I was taught at university". Yet from a design perspective it also introduces challenges.

Typically development teams would ideally like to achieve this level of reuse. But as enterprise's resources are often limited this often means only one application can be specified, designed and built at a time. So in the process of building BTFs for one application with the goal of reusing them in another application yet to be specified, how do you go about designing and building the BTFs for requirements of the future system that has not yet been detailed? Should you extend all your BTFs for example that allow users to edit data with an optional readOnly parameter? But what if only half your BTFs end up getting reused, is adding that parameter by default an overkill in this regards?

Considerably build, versioning and dependency management issues become even more complicated once we establish this level of reuse too. If two master workspaces reuse the same BTF workspace, but require different versions of the BTF workspace, and multiply that by 10s if not 100s of BTF workspaces, keeping track of all of this and building your applications in the right order may become extremely complex and time consuming.

Reuse may feel like a programmer's nirvana, but it can be taken too far. Ultimately every flexibility has a cost associated with it, and it's up to architects and developers to take the opportunity into account rather than following an ideal blindly which can have significant costs associated with it.

The Cylinder Architectural Pattern

Stepping back from the Two-for-One architectural pattern, there is another interesting issue that wasn't articulated in the Sum-of-the-Parts pattern.

A key changing part for any brand new ADF system written from the ground up is the ADF Business Components. As our developers working on their separate BTF workspaces may require numerous changes to the ADF Business Components to support their own needs, the knock on effect is that separate BTF workspaces may also be impacted by these continuous changes to the ADF BC layer from the Common workspace. Can we propose an architectural pattern that will isolate teams from changes to what is likely the most fluid part of our shared components?

The *Cylinder Architectural Pattern* as shown in Figure 5 proposes a distinct change from the Sum-of-the-Parts Pattern in the fact that each BTF workspace gets its own ADF BC Model project:

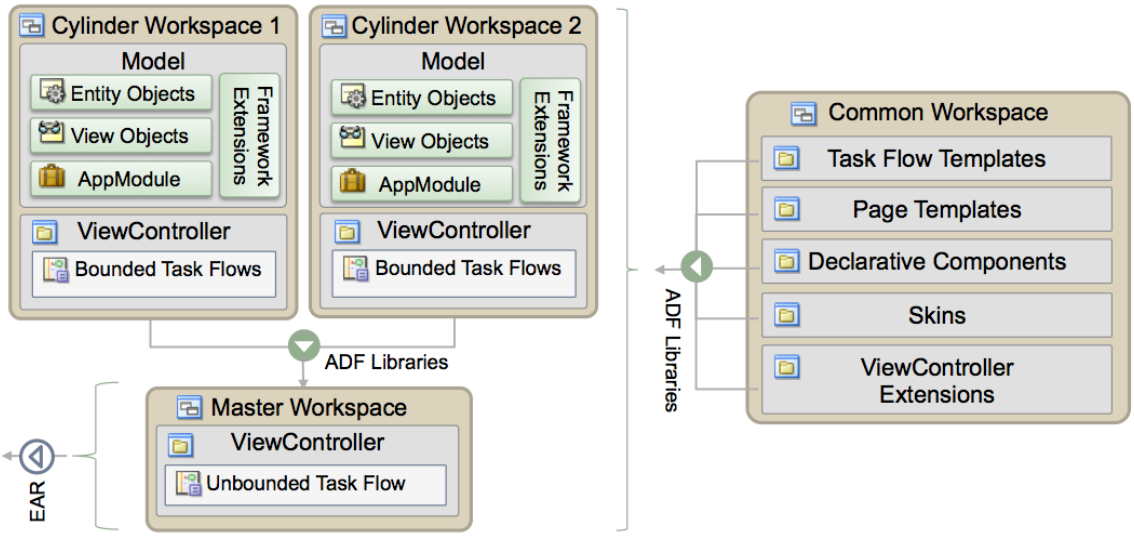

Figure 5 - The Cylinder Pattern

The clear advantage of this pattern is that each BTF workspace developer is now isolated from the more rapidly changing ADF BC Model changes required by other teams. By having their own copy of ADF Business Components they can control the tempo of changes and not be severely impacted by the acts of other team members.

Of course the major disadvantage of this pattern is we now have the potential for many duplicated pieces of code. If two workspaces require a Customer Entity Object, and a new column is added to the relating Customer table, it is up to each BTF workspace developer to keep their EO up to date (with all the potential risks they wont).

Ultimately this all about a trade off. Stretching the separation of our solutions so they nearly become isolated cylinders that don't rely on each other and affect each other with changes, with the expense to reuse our business components and avoid maintenance issues.

In considering this pattern it's worth noting that a less-changing reusable artefacts such as task flow templates, page templates, declarative components, skins and other utility classes are still good

candidates for a common workspace. As these may likely change fairly rapidly initially, but then settle down and likely change rarely over a project, it's still good practice to place them into a separate shared resource. Of course if they become a rapidly changing construct too, then potentially the same consideration should be made, can we move some of these shared artefacts to each BTF workspace, and at one cost to maintenance?

A final point to make is this architectural pattern will not protect you from a rapidly changing database model. Skilled database developers will know a level of stability in your database model is essential for not disrupting your dependent applications, including but not limited to just ADF systems. Employing a seasoned database modeller is well worth the cost versus the disruption an unstable database model can cause during a software build.

The Pillar Pattern

There is another interesting challenge that has yet to be addressed by any architectural pattern seen to date. In the previous architectural patterns the end deployment artefact is a single EAR. Arguably WebLogic Server shared libraries can be utilized in deploying ADF library JARs to be loaded by the main EAR, but the end result is still a single application running on the server. So what happens when that application grows so large that in grows beyond the capabilities of a single WLS or Java EE server to run?

This might seem unlikely that you can build an application to this scale, but certainly some programming teams do hit these limits. If you consider Oracle's own Fusion Applications which is made up of numerous modules, HR, Financials, Incentive Compensation and is still growing to this day, it simply is an application that is too large to run on a single Java EE server with ultimately limited JVM, CPU and memory resources. So how do you solve this?

The *Pillar Architectural Pattern* as extension to the Cylinder Pattern seen earlier attempts to solve this problem as described in Figure 6:

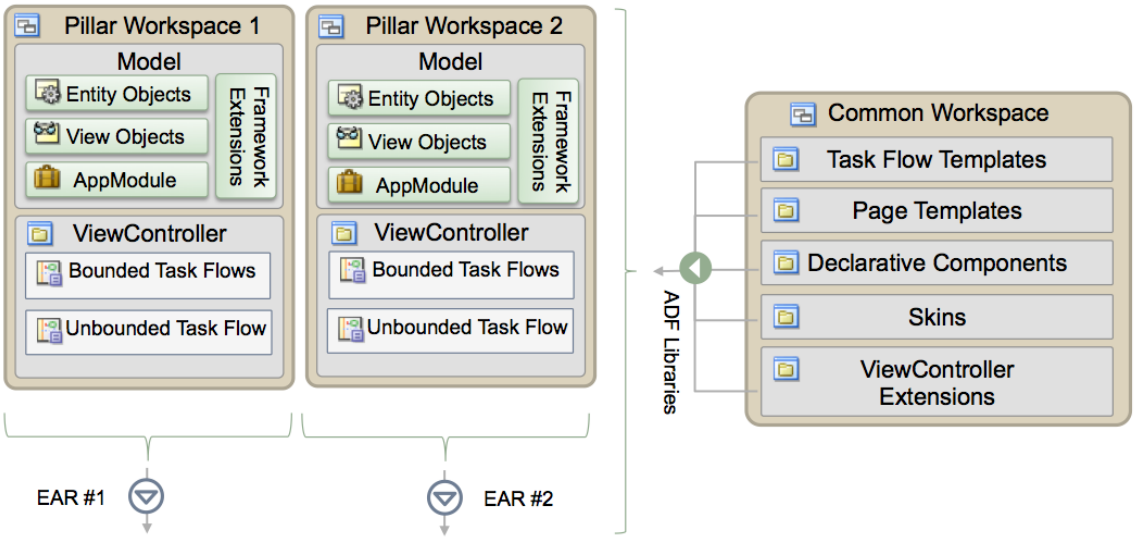

Figure 6 - The Pillar Pattern

The key difference between this pattern and the previous patterns is that each Pillar is a deployed application in its own right. Rather than a final single master/composite application workspace taking all the BTFs from each cylinder and presenting a single unified application through a single

Unbounded Task Flow, each Pillar has its own UTF and is deployed and treated as a standalone application it's own right.

The predominate advantage of this architectural patterns is you can now shift each pillar to a separate Java EE node where it will get its own dedicated resources.

Yet such an architecture introduces a number of additional challenges that were not present in a single unified application. Technically the logical application is separate applications, deployed separately, accessed separately, with different memory models, separated user state, separate authentication schemes. This will throw up issues like:

1. The user will not want to login to every single application so a Single Sign On mechanism is an imperative.

2. The user will expect the logical application to be a single logical application so will have expectations of a single application chrome/shell, a single menu model, a unified look and feel. You will need to unify your applications so they feel like one application.

3. And the user will when, say, working in the Financials pillar editing one customer invoice, will expect when navigation to the Stock Order system to look at the order of that customer, that the customer details are shared across the applications. They wont want to have to copy and paste the customer ID between systems. So a mechanism to share state between pillars is required too.

All of these challenges can be overcome but as can be seen it does add additional complexity to a solution which may be beyond the skills and experience of many smaller and less advanced development teams. Is the Pillar Pattern for you?

Conclusion

As Oracle's Application Development Framework continues to mature there is a distinct opportunity for ADF development teams to look beyond the how-do-I-get-this-selectOneChoice-to-work to a broader concern on how to make best use of this framework to limit development time. Through the opportunities of reuse ADF provides an extremely flexible solution for achieving this. Yet it is important to have an eye to the different architectural patterns and their pros and cons versus your own requirements early on such that the wrong architecture solution isn't chosen, or even worse no architecture what so ever.

For Oracle customers interested in learning more about ADF architecture concepts, you're encouraged to pursue the ADF Architecture TV channel, a set of YouTube videos focusing on the design & architecture concerns of ADF systems: http://bit.ly/adftvsub

Contact :

Name: Chris Muir & Frank Nimphius

Firma: Oracle Corporation

Internet: www.oracle.com